Microarchitecture Optimizations for Exploiting Memory-Level Parallelism

Yuan Chou, Brian Fahs and Santosh Abraham
Architecture and Advanced Development
Processor and Network Products
Sun Microsystems
Sunnyvale, CA 94086
{yuan.chou,brian.fahs,santosh.abraham}@sun.com

Abstract

The performance of memory-bound commercial applications such as databases is limited by increasing memory latencies. In this paper, we show that exploiting memory-level parallelism (MLP) is an effective approach for improving the performance of these applications and that microarchitecture has a profound impact on achievable MLP. Using the epoch model of MLP, we reason how traditional microarchitecture features such as out-of-order issue and state-of-the-art microarchitecture techniques such as runahead execution affect MLP. Simulation results show that a moderately aggressive out-of-order issue processor improves MLP over an in-order issue processor by 12-30%, and that aggressive handling of loads, branches and serializing instructions is needed to attain the full benefits of large out-of-order instruction windows. The results also show that a processor's issue window and reorder buffer should be decoupled to exploit MLP more efficiently. In addition, we demonstrate that runahead execution is highly effective in enhancing MLP, potentially improving the MLP of the database workload by 82% and its overall performance by 60%. Finally, our limit study shows that there is considerable headroom in improving MLP and overall performance by implementing effective instruction prefetching, more accurate branch prediction and better value prediction in addition to runahead execution.

1 Introduction

Over the past two decades, instruction-level parallelism (ILP) has been a primary focus of computer architecture research and a variety of microarchitecture techniques to exploit ILP such as pipelining, VLIW and superscalar issue have been developed and refined. Further, many speculation techniques to enhance ILP such as branch prediction and data speculation have been studied in academia and adopted in commercial microprocessors. These advances enable current microprocessors to effectively utilize deep multiple-issue pipelines in some classes of applications, such as media processing and scientific floating-point intensive applications.

On the other hand, the performance of commercial applications such as databases are dominated by the frequency and cost of memory accesses. These applications typically have large instruction and data footprints that do not fit in a processor's on-chip caches, thereby requiring frequent accesses to off-chip caches or memory [1, 2]. Further, in contrast to many media and scientific applications, these applications exhibit control- and data-dependent irregular patterns in their memory accesses that are not amenable to conventional hardware or software prefetching.

In today's database workloads, almost two-thirds of execution time is spent in off-chip accesses [3]. As numerous papers have pointed out, the latency for these off-chip accesses in processor cycles continues to grow because the rapid improvements in processor clock frequencies have outpaced the improvements in memory and interconnect speeds. Some have termed this trend the impending "memory wall" problem [4]. For memory-bound workloads, an inordinate amount of ILP must be uncovered and exploited in order to hide the high latency of their off-chip accesses. Even if this much ILP exists in these applications, exploiting them using conventional techniques such as out-of-order execution requires extremely large instruction windows that are very difficult to implement. For these applications, a promising alternative is to exploit memory-level parallelism (MLP) by overlapping multiple off-chip accesses. To illustrate the potential of MLP as a performance lever, if a memory-bound application spends two-thirds of its execution time in off-chip accesses, doubling the MLP can halve the time spent in these accesses and potentially improve performance by 50%.

In this paper, we examine the impact of high-level microarchitecture choices on MLP, and identify and evaluate promising approaches for exploiting MLP. We perform our study using three commercial workloads that collectively represent all three tiers of the corporate datacenter. Since the term MLP has hitherto been used informally in the literature [5, 6, 7], we define MLP formally and relate it to overall performance. To assist our study, we use the epoch model, which is a simple but powerful model to reason about MLP and how microarchitecture features impact MLP. This epoch model is implemented in MLPsim, our MLP simulator that has been thoroughly validated against a cycle-accurate simulator. Using MLPsim, we show that serializing instructions (e.g. CASA in the SPARC ISA) have a significant impact on achievable MLP in conventional out-of-order processors. We then study the impact of state-of-the-art microarchitecture features such as runahead execution [8, 9] and value prediction [10, 11, 12, 18] on MLP and

1063-6897/04 $20.00 (c) 2004 IEEE

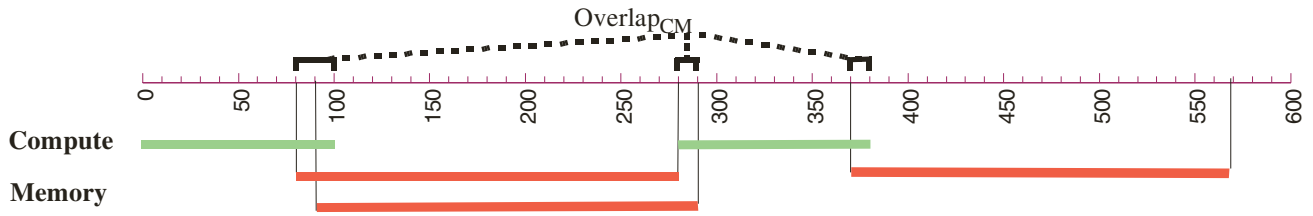

Figure 1: Example of Compute-Memory and Memory-Memory Overlap.

demonstrate that runahead execution can significantly improve MLP. Lastly, we perform a limit study on achievable MLP to reveal profitable research directions for improving MLP. Beyond merely presenting performance numbers, our foremost objective is to reveal insights into the key microarchitecture factors inhibiting MLP.

The rest of this paper is organized as follows. Section 2 defines MLP, relates it to overall processor performance and describes how it can be measured. Section 3 presents the epoch model of MLP and uses this model to explain how microarchitecture features affect MLP. Section 4 describes our experimental methodology, including our MLPsim tool, while Section 5 presents our experimental results. Section 6 describes related work and Section 7 summarizes the contributions and conclusions of this paper.

2 MLP and Overall Performance

2.1 Definition

We define instantaneous MLP, MLP(t) as *the number of useful long-latency off-chip accesses outstanding at cycle t*. We define average MLP, MLP as *the average number of useful long-latency off-chip accesses outstanding when there is at least one such access outstanding*. MLP can also be derived by averaging MLP(t) over all the cycles when there is at least one access outstanding.

In our definition, off-chip accesses include instruction fetches, loads and hardware/software prefetches. We use the qualifier 'long-latency' for off-chip accesses because some current processors have off-chip level-two (L2) caches that have relatively low latency. We believe that as on-chip L2 and level-three (L3) caches become more common due to increasing transistor density, future off-chip caches will be large and have high latencies, so the qualifier 'long latency' will be eventually unnecessary. For the rest of this paper, we refer to long latency off-chip accesses simply as off-chip accesses. We also use the qualifier 'useful' so that we do not count useless accesses that would otherwise inflate MLP without presenting any benefit. More specifically, useful accesses refer to those made non-speculatively as well as to those made speculatively that bring in data that is accessed by subsequent non-speculative instructions. Note that hardware/software prefetches contribute to MLP only if they bring in data that is used by either a subsequent non-speculative load or a subsequent non-speculative instruction fetch.

In order to measure MLP in a cycle-accurate processor simulator, we first measure MLP(t) every cycle by counting the number of useful off-chip accesses outstanding that cycle. We then derive MLP by averaging the MLP(t) of those cycles where MLP(t) is non-zero.

2.2 Relating MLP to Overall Performance

Assuming a constant off-chip access latency, we can relate MLP to overall execution time using the following equation:

$$Cycles = Cycles_{perf}(1 - Overlap_{CM}) + \frac{NumMisses \times MissPenalty}{MLP}$$

where *Cycles* is total execution cycles, $Cycles_{perf}$ is number of execution cycles if the furthest on-chip cache is perfect, $Overlap_{CM}$ is the fractional overlap of compute cycles with off-chip cycles, *NumMisses* is the number of off-chip accesses, *MissPenalty* is the latency of each off-chip access, and *MLP* is the average memory-level parallelism.

In Figure 1, we show the timing of an out-of-order processor with a memory latency of 200 cycles. The light bars show periods of active instruction issue while the dark bars show three off-chip accesses. The first term in the equation captures the compute-only cycles, i.e. cycles with *only* a light bar and the second term captures the cycles covered by *at least* one off-chip access, i.e. a dark bar. In this example, the first term covers cycles 0-80 and cycles 290-370 while the second term covers cycles 80-290 and cycles 370-570. The equation parameters are: Cycles = 570, $Cycles_{perf}$ = 200, NumMiss = 3, MissPenalty = 200, $Overlap_{CM}$ is (20 +10 + 10)/200 = 0.2, and MLP = 1.463.

Alternately, we can express the equation in terms of CPI:

$$CPI = CPI_{perf}(1 - Overlap_{CM}) + \frac{MissRate \times MissPenalty}{MLP}$$

where *CPI* is the overall CPI and CPI_{perf} is the perfect on-chip cache CPI and *MissRate* is the miss rate per instruction of the furthest on-chip cache. The first term of this equation is $CPI_{on\text{-}chip}$ and

Benchmark	Off-Chip Latency	CPI	$CPI_{on\text{-}chip}$	$CPI_{off\text{-}chip}$	L2 Miss Rate (per 100 insts)	MLP	$Overlap_{CM}$
Database	200	2.44	1.47	0.97	0.84	1.33	0.20
Database	1000	7.28	1.47	5.81	0.84	1.38	0.18
SPECjbb2000	200	1.45	1.16	0.29	0.19	1.13	0.04
SPECjbb2000	1000	2.80	1.16	1.64	0.19	1.14	0.04
SPECweb99	200	1.73	1.62	0.11	0.09	1.25	0.02
SPECweb99	1000	2.30	1.62	0.68	0.09	1.29	0.00

Table 1: Measurements of On-Chip and Off-Chip Components of CPI.

the second term is $CPI_{off\text{-}chip}$.

We can use a cycle-accurate simulator to measure CPI_{perf} and CPI. In the first run, we measure CPI_{perf} by running the simulator using a perfect on-chip L2 cache (assuming the L2 cache is the furthest on-chip cache). In the second run using a realistic L2 cache, we measure overall CPI and MLP. $Overlap_{CM}$ is then derived from Eq. (2) by substituting for the other known parameters. Table 1 shows the values of CPI, $CPI_{on\text{-}chip}$, $CPI_{off\text{-}chip}$, $Overlap_{CM}$ and MLP for three commercial applications as off-chip access latency is increased from 200 cycles to 1000 cycles. The out-of-order processor configuration simulated corresponds to that described in Section 5.1.

At the off-chip latency of 1000 cycles, $CPI_{off\text{-}chip}$ is more than 3x $CPI_{on\text{-}chip}$ for the database workload and about 1.5x $CPI_{on\text{-}chip}$ for SPECjbb2000. For SPECweb99, the $CPI_{off\text{-}chip}$ is not that large relative to $CPI_{on\text{-}chip}$ but is still substantial. A memory latency of 1000 cycles may appear to be excessively large, but is consistent with a reasonable future system with multiple processor chips operating at 5 GHz and a 200 ns off-chip latency. Conventional out-of-order techniques are not effective in overlapping memory cycles and compute cycles as indicated by the modest $Overlap_{CM}$, though they are somewhat effective at issuing multiple off-chip accesses in parallel as indicated by the measured MLP that ranges from 1.14 through 1.38. For these workloads, exploiting ILP in a manner that address only $CPI_{on\text{-}chip}$ is not going to improve overall CPI substantially. In contrast, increasing MLP is a powerful performance lever that can lead to significant improvements in off-chip CPI as well as overall CPI.

2.3 Clustering of Off-Chip Accesses

With the default processor configuration described in Section 5.1, the miss rate (per 100 instructions) for the database workload, SPECjbb2000 and SPECweb99 is 0.84, 0.19 and 0.09 respectively. Thus, the average distance between two consecutive off-chip accesses is 119, 526, 1111 instructions respectively. With such large inter-miss distances, particularly in the case of SPECjbb2000 and SPECweb99, one might expect that an instruction issue window of size 64 is unlikely to capture two or more misses and therefore MLP will be close to 1. On the contrary, Table 1 shows that the MLP for SPECweb99 is almost 1.3. In order to understand this apparent inconsistency, we plot the cumulative probability of encountering another off-chip access within a certain number of dynamic instructions. In Figure 2, the three thinner curves assume an uniform distribution of inter-miss distances and are derived from the average inter-miss distances observed for these workloads. The three thicker curves are the observed distribution of cumulative probability versus inter-miss distances. As indicated by the divergence between the observed and uniform distribution curves, the observed inter-miss distributions are extremely clustered for SPECweb99 and SpecJBB2000. These clustered inter-miss distributions suggest that exploiting MLP is indeed a viable proposition.

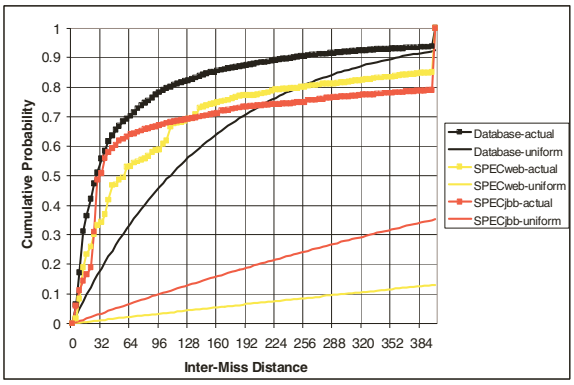

Figure 2: Clustering of Misses.

3 Microarchitecture Impact on MLP

In this section, we present our epoch model and use this model to reason how microarchitecture features impact achievable MLP. To aid our explanation, we define the following terms: a *missing load (Dmiss)* is a load that requires an off-chip access, a *missing prefetch (Pmiss)* is a useful read prefetch that requires an off-chip access, and a *missing instruction fetch (Imiss)* is an instruction fetch that requires an off-chip access. Also, when we say that two off-chip accesses can be overlapped, we imply that there are no true data dependences between them.

In this section and for the rest of this paper, we assume that the processor's load and store buffers are large enough so that instruction decode/dispatch is never stalled because of insufficient load or store buffer entries.

3.1 Epoch Model

As off-chip access latencies increase, on-chip computation latencies separating overlappable off-chip accesses become increasingly insignificant relative to off-chip access latencies. By on-chip computation, we refer to data computations, address computations, as well as instruction fetches and loads that hit in the on-chip caches. In such a situation, illustrated in Figure 3, overlappable off-chip accesses appear to issue and complete at the same time, and instruction execution tends to separate itself into recurring periods of on-chip computations and off-chip accesses.

We call each such period of on-chip computations followed by off-chip access(es) an epoch. More precisely, an *epoch* is a time slice of program execution starting from the end of the previous epoch through the first off-chip access and extending to the cycle when this first access has completed. We define the *epoch trigger* as the instruction that is responsible for the first off-chip access of an epoch. Within an epoch, all overlappable off-chip accesses are

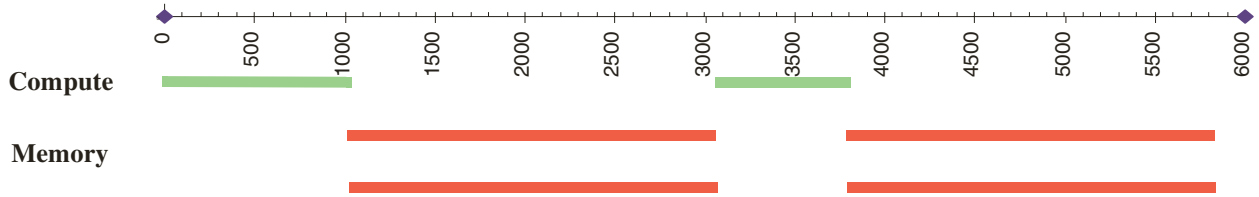

Figure 3. Example where Off-Chip Latency is Much Greater than On-Chip Latencies.

assumed to issue and complete at the same time.

To understand how microarchitecture constrains MLP, it is necessary to understand how it affects which off-chip accesses can be issued in an epoch. This in turn requires us to understand which instructions in the processor's dynamic instruction stream (DIS) can be issued in a given epoch. For this purpose, we define an *epoch set* as the set of instructions from the DIS that can be executed in an epoch. We can then partition the DIS into epoch sets such that all the instructions in epoch set *i* execute in the *i*th epoch. MLP is then simply the ratio of the total number of off-chip accesses to the number of epoch sets.

3.2 Primary Limiters of MLP in Out-of-Order Issue Processors

Out-of-order processors issue independent instructions from a window of dynamic instructions. A *window termination* condition is a condition that prevents the processor from executing any subsequent instructions in the DIS till all earlier off-chip accesses are resolved. Thus, an epoch set shall not include any more instructions after a window termination. Specifically, an epoch set is a subset of the instructions in the DIS before a window termination condition, comprising those instructions that are not dependent on other off-chip accesses in this epoch and that are not already part of earlier epoch sets. A set of microarchitecture choices imposes an associated set of window termination conditions. In the absence of window termination conditions, the number of epoch sets and consequently MLP is determined solely by data dependencies between missing loads.

For out-of-order processors, we identify four key window termination conditions: 1) issue window size and reorder buffer size limitations, 2) serializing instructions, 3) instruction fetch misses and 4) unresolvable branch mispredictions.

We use Examples 1-3 to illustrate the effects of these window termination conditions. In these examples, *Dmiss* indicates that the instruction is a missing load and *Mispred* indicates that it is a mispredicted branch. In each case, we list the instructions in each epoch set, where instructions in bold resulted in a useful off-chip access and instructions in italics (e.g. i2 in Example 3) were fetched but not executed in that epoch.

3.2.1 Issue Window Size and ROB Size

Processors that implement out-of-order instruction issue typically have a hardware structure that remembers the fetch order of the instructions in order to provide in-order retirement. This structure is usually named the *reorder buffer* (ROB). They also typically have a hardware structure that stores the instructions that have been renamed but not yet issued. We call this structure the *issue window* although there are other commonly used names for it (e.g. reservation station). When a missing load epoch trigger becomes the oldest entry in the ROB, it will hold up further instruction retirement until its data returns. As a result, the ROB and/or the issue window soon becomes full and no more missing loads or missing prefetches can be overlapped. Therefore, both the ROB size and the issue window size have a potentially large impact on MLP.

In Example 1, we assume that the issue window/ROB size of the processor is four. In the first case, the window begins at i1, a missing load, and terminates at instruction i4 because of the window size constraint of four. Instruction i1 leaves the ROB only after the miss is processed. Therefore, i5 cannot enter the ROB until i1 is retired. Data dependencies prevent instructions i2 and i3 from being issued in parallel with i1. Thus, the first epoch set contains i1 and i4 and the second epoch set contains the remaining instructions.

3.2.2 Serializing Instructions

Most instruction set architectures (ISAs) provide instructions for implementing synchronizing primitives used for concurrency control as well as instructions for explicitly enforcing memory ordering. In the SPARC ISA [13], examples of the former include the CASA and LDSTUB instructions, and an example of the latter is the MEMBAR instruction. For this paper, we call these instructions serializing instructions because a straightforward implementation usually requires the processor pipeline to drain before these instructions can be issued. Draining the pipeline in turn requires all the instructions older than the serializing instruction to be architecturally committed. Therefore, missing loads and missing prefetches after the serializing instruction cannot overlap with missing loads and missing prefetches before it and consequently MLP is reduced.

In Example 2, we illustrate the effects of a MEMBAR serializing instruction. The window starts at i1 and terminates at the serializing instruction i2. The first epoch set only includes these instructions and the next epoch contains the remaining instructions.

Although it is beyond the scope of this paper, we note that very aggressive and speculative implementations of these serializing instructions are possible that eliminate the need for draining the pipeline except in rare cases of misspeculations. We also note that these serializing instructions can be fairly prevalent. For example, CASA instructions are used for Java object locking and make up more than 0.6% of the total dynamic instruction count in SPECjbb2000.

3.2.3 Instruction Fetch Misses

An instruction fetch that misses the on-chip caches terminates

Inst#	Instruction	Type
i1	load 0(r1)->r2	*Dmiss*
i2	add r2,r3->r4	
i3	load (r4)->r5	*Dmiss*
i4	add r0,r1->r2	
i5	load (r7)->r8	*Dmiss*

Epoch Sets =
{**i1**, i4}, {i2, **i3**, **i5**}
MLP = (1+2)/2 = 1.5

Example 1

Inst#	Instruction	Type
i1	load (r1)->r2	*Dmiss*
i2	membar	
i3	add r2,r3->r4	
i4	load (r4)->r5	*Dmiss*
i5	load (r7)->r8	*Dmiss*

Epoch Sets =
{**i1**, i2}, {i3, **i4**, **i5**}
MLP = (1+2)/2 = 1.5

Example 2

Inst#	Instruction	Type
i1	load (r1)->r2	*Dmiss*
i2	add r2,r3->r4	*Imiss*
i3	load (r4)->r5	*Dmiss*
i4	beq r5,0, tgt	*Mispred*
i5	load (r7)->r8	*Dmiss*

Epoch Sets =
{**i1**, ***i2***}, {i2, **i3**}, {i4, **i5**}
MLP = (2+1+1)/3 = 1.33

Example 3

Inst#	Instruction	Type
i1	load 8(r1)->r2	*Dmiss*
i2	load 0(r2)->r3	*Dmiss*
i3	load 108(r1)->r4	*Dmiss*
i4	store r5->0(r3)	
i5	load 388(r1)->r6	*Dmiss*

1. Epoch Sets = {**i1**}, {**i2**, **i3**}, {i4, **i5**}
2. Epoch Sets = {**i1**, **i3**}, {**i2**}, {i4, **i5**}
3. Epoch Sets = {**i1**, **i3**, **i5**}, {**i2**}, {i4}

Example 4

Inst#	Instruction	Type
i1	load 8(r1)->r2	*Dmiss*
i2	beq r2, 1, 0x1100	
i3	beq r1, 1, 0x11ff	*Mispred*
i4	load 108(r1)->r4	*Dmiss*

1. Epoch Sets = {**i1**}, {i2, i3, **i4**}
2. Epoch Sets = {**i1**, i3, **i4**}, {i2}

Example 5

a window because it prevents any further instructions in the DIS from entering the processor's issue window. In Example 3, i2 is an instruction miss and terminates the window begun by i1. Only the instruction fetch part of i2 is completed in the first epoch. In the next epoch, i2 is executed along with other instructions.

3.2.4 Unresolvable Branch Mispredictions

Following a branch misprediction, the processor fetches and executes instruction along the wrong path until the branch is resolved. In general, the wrong-path instructions are not present along the correct path and any off-chip accesses initiated along the wrong-path are not useful and do not contribute to MLP. Of course, if the wrong-path converges with the correct path, off-chip accesses uncovered by the wrong path may be useful (if their addresses are the same on both paths). Branches that are dependent on on-chip computations quickly resolve themselves. However, mispredicted branches that are dependent on a missing load in the current epoch are *unresolvable* and cause the processor to continue along the wrong path until the end of the epoch. In Example 3, i3 is a missing load and the mispredicted branch i4 is unresolvable because it is dependent on i3. Thus, i5 cannot be fetched until after i3 and i4 complete. In effect, i4 terminated the window.

Since the accuracy of the branch predictor has a significant impact on window terminations and MLP, it may be desirable to design special branch predictors for unresolvable branches. Since the resolution latency is proportional to off-chip latency, these branch predictors can be very effective even if they are slow.

3.3 In-order Processors

In-order processors have very limited capabilities for overlapping multiple off-chip accesses. The window termination conditions such as those for serializing instructions, instruction fetch misses, and unresolvable branch mispredictions apply to in-order processors as well. In-order processors may generally be classified as stall-on-miss or stall-on-use. In a stall-on-miss in-order processor, the processor stalls instruction issue when a load misses the data cache. Thus, a missing load starts and terminates a window. However missing useful hardware/software prefetches and missing instruction fetches may be overlapped among themselves or with a missing load. The MLP is therefore usually somewhat higher than unity. In a stall-on-use processor, the processor stalls instruction issue when the data of a missing load is used by a subsequent instruction. Thus, the missing load starts a window and its first dependent instruction terminates the window. In contrast to a stall-on-miss processor, additional load misses between a missing load and its use may be overlapped. Therefore, the MLP of a stall-on-use processor is slightly higher than that of a stall-on-miss processor.

3.4 Issue policies and MLP

Out-of-order issue processors usually allow instructions from different instruction classes to be issued out-of-order freely with respect to each other but may place specific constraints on the issue of instructions within the same instruction class.

3.4.1 Load Issue Policy

In the case of loads, processors typically implement one of the following three issue policies:

1. Loads are issued in-order w.r.t. other loads and stores. This allows a straightforward implementation of the processor consistency memory model.

2. Loads are issued out-of-order w.r.t. other loads but wait for address resolution of all earlier stores.

3. Loads are issued out-of-order w.r.t. other loads and do not wait for address resolution of earlier stores.

Example 4 illustrates how the choice of load issue policy affects MLP and lists the epoch sets for each of the three cases. If the first policy is implemented, no other missing loads can be overlapped with i1 because i2, which has a dependence on i1, prevents subsequent loads from being executed. If the second policy is implemented, i3 can be overlapped with i1 while i5 cannot do so because it has to wait for the address of store i4 to be resolved. Unfortunately, store i4 in turn has a dependence on i2. If the third policy is implemented, both i3 and i5 can be overlapped with i1.

3.4.2 Branch Issue Policy

Out-of-order issue processors typically allow branches to execute out-of-order with respect to other instruction types but may constrain the issue ordering among branches. Two policies are possible:

1. Branches are issued in-order w.r.t. other branches.

2. Branches are issued out-of-order w.r.t. to other branches.

Example 5 illustrates how the choice of branch issue policy affects MLP. If the first policy is implemented, i4 cannot be overlapped with i1. Since i2 has a dependence on i1, it cannot be executed in the same epoch. As a result, branch i3, which must be issued in-order, cannot resolve and the processor continues executing on the wrong path until the end of the epoch. In contrast, if the second policy is implemented, i4 can be overlapped with i1.

3.5 Runahead Execution

After the missing load epoch trigger becomes the oldest entry in the ROB, no more instructions can be retired until the missing load's data returns. In the meantime, the ROB and the issue window

fills up quickly, effectively stalling the processor. In runahead execution [8, 9], when the missing load epoch trigger becomes the oldest entry in the ROB, the processor checkpoints the register files and enters runahead execution mode. In this mode, all missing loads (including the epoch trigger) are converted to prefetches and therefore do not stall retirement. Any subsequent instructions that are dependent on these missing loads are simply skipped. Also, stores do not update architected state. When the data of the missing load epoch trigger returns, the processor flushes its pipeline, restores the register checkpoint and re-enters normal execution mode.

Because runahead execution is purely speculative, it does not have to obey the serialization constraints of serializing instructions. Therefore, runahead execution increases MLP because it removes all window termination conditions except instruction fetch misses and unresolvable branch mispredictions.

3.6 Value Prediction

Value prediction [10, 11, 12, 18] increases MLP because the correct prediction of a missing load's value allows subsequent dependent missing loads to issue in the same epoch. To improve MLP, we only need to predict the values of missing loads instead of the values of all loads or all result-producing instructions. This can drastically reduce the size of the value predictor.

4 Experimental Methodology

In this section, we describe MLPsim, the tool we developed for our MLP study, as well as the benchmarks we used for our experiments.

4.1 MLPsim

MLPsim is our MLP simulator that implements the epoch MLP model. It reads in an instruction trace and a set of microarchitecture parameters, and outputs MLP and epoch statistics by partitioning the instruction trace into epoch sets. It does so by tracking register and memory dependences between instructions, modeling the sizes of three key hardware structures: fetch buffer, issue window and reorder buffer, and applying the window termination conditions associated with the microarchitecture parameters specified.

MLPsim does not need to model any on-chip computations such as cache misses that do not require an off-chip access or branch mispredictions that are not dependent on a missing load. Neither does it need to model the timing of these computations (which means that it does not need to model instruction latencies, fetch bandwidth, issue bandwidth, number and types of function units etc.). In fact, it does not even need to model the timing of off-chip accesses since the epoch model assumes that off-chip accesses that are overlappable in an epoch issue and complete at the same time. By taking advantages of the simplifications enabled by the epoch model, MLPsim is simple, small and easy to verify. As the results in Section 5.2 demonstrate, the MLP results from MLPsim and our cycle-accurate simulator match extremely closely, especially as off-chip access latency reaches 500 cycles and beyond. Because of its simplicity and accuracy, MLPsim is a convenient tool for exploring and prototyping new microarchitecture features for enhancing MLP. MLPsim can also be used as a simple processor model that accurately estimates the clustering of off-chip accesses in simulation-based queueing models of memory and system interconnects.

4.2 Benchmarks

We used three commercial workloads in our study: a database workload, SPECjbb2000 [14] and SPECweb99 [14]. Collectively, they cover all three tiers of a corporate datacenter. Due to disclosure agreements, we are not able to elaborate on the details of our database workload. SPECjbb2000 is a server-side Java benchmark that emphasizes business logic and object manipulation, the middle tier of a 3-tier system, while SPECweb99 evaluates the performance of web servers.

The binaries used to generate our traces are highly optimized ones that are used by our company for reporting benchmark results. The traces were collected when the workloads were warmed and running in steady state and they were meticulously validated against hardware counter statistics.

For all our simulations, we used the first 50M instructions in the trace to warm the caches and the next 100M instructions to collect statistics. All three workloads we used are transaction-oriented and do not exhibit phase changes. We have carefully validated that 50M instructions are sufficient for warming the L2 cache and 100M instructions are sufficient for collecting a representative transaction mix that enables accurate statistics collection.

5 Experimental Results

In this section, we first compare the results of MLPsim with our cycle-accurate simulator and then show the results of using MLPsim to quantify the impact of different microarchitecture parameters and features on MLP.

5.1 Default Processor Configuration

Unless stated otherwise, our experiments assume the following default processor configuration:

- 32KB 4-way set-associative, 64B line size L1 instruction and data caches
- 2MB 4-way set associative, 64B line size shared L2 cache
- no L3 cache
- 2K entry shared TLB
- 64K entry gshare branch predictor, 16K entry branch target buffer, 16 entry return address stack
- 32 entry fetch buffer, 64 entry instruction issue window, 64 entry reorder buffer
- infinite load buffer and store buffer
- out-of-order instructions issue; loads issue out-of-order with respect to other loads and stores and speculate past earlier stores with unresolved addresses, branches issue in-order with respect to other branches (i.e. issue configuration C in Table 2)

5.2 MLPsim Validation

We first validate the MLP results produced by MLPsim against those produced by our cycle-accurate simulator, a simulator that was designed to explore microarchitecture ideas for future SPARC processors for our company.

For our validation, we modeled three different instruction window/ROB sizes and three different issue constraints configurations. For this experiment, we set the instruction window size to be equal to the ROB size. The issue constraints configurations correspond to the first three configurations shown in Table 2. We did not use configurations D and E because our cycle-accurate simulator cannot

Issue Configuration	Load Issue (w.r.t other loads)	Branch Issue (w.r.t. other branches)	Serializing Instructions
A	in-order	in-order	serializing
B	out-of-order, wait for earlier store addresses to resolve	in-order	serializing
C	out-of-order, speculate past earlier stores	in-order	serializing
D	out-of-order, speculate past earlier stores	out-of-order	serializing
E	out-of-order, speculate past earlier stores	out-of-order	non-serializing

Table 2: Configurations of Issue Constraints.

ROB/Issue Window Size	Issue Config	Database				SPECjbb2000				SPECweb99			
		CycleSim 200	CycleSim 500	CycleSim 1000	MLPsim	CycleSim 200	CycleSim 500	CycleSim 1000	MLPsim	CycleSim 200	CycleSim 500	CycleSim 1000	MLPsim
32	A	1.19	1.20	1.20	1.21	1.10	1.10	1.10	1.10	1.19	1.20	1.21	1.20
	B	1.20	1.22	1.22	1.23	1.11	1.11	1.11	1.10	1.19	1.20	1.21	1.20
	C	1.25	1.26	1.27	1.27	1.11	1.12	1.12	1.11	1.21	1.23	1.23	1.22
64	A	1.22	1.22	1.24	1.25	1.10	1.10	1.10	1.10	1.22	1.24	1.24	1.23
	B	1.24	1.27	1.28	1.28	1.13	1.13	1.13	1.13	1.22	1.24	1.25	1.24
	C	1.33	1.37	1.38	1.38	1.13	1.14	1.14	1.13	1.25	1.28	1.29	1.28
128	A	1.24	1.26	1.27	1.28	1.13	1.14	1.15	1.15	1.23	1.25	1.26	1.25
	B	1.27	1.30	1.31	1.32	1.16	1.18	1.19	1.19	1.23	1.25	1.26	1.25
	C	1.39	1.44	1.46	1.47	1.17	1.19	1.20	1.19	1.27	1.30	1.31	1.31

Table 3: Comparison of MLP numbers by MLPsim and Cycle-Accurate Simulator.

simulate out-of-order branch execution.

Table 3 compares the MLP results from MLPsim and our cycle-accurate simulator. The results show that despite MLPsim's simplicity, its results match very well with the much more complex cycle-accurate simulator. In fact, when memory latency is 1000 clocks, their results are almost identical. These results give us confidence not only in the correctness and robustness of MLPsim but also in the validity of the epoch model and the completeness of the "rules" described in Section 3.

We now validate the second equation of Section 2.3. We use this equation to estimate the CPI of a configuration by substituting its MLP and MissRate measured by MLPsim and its CPI_{perf} and $Overlap_{CM}$ measured by the cycle-accurate simulator (shown in **bold** in Table 4), and compare this to its CPI measured by the cycle-accurate simulator (shown in *italics* in Table 4).

We also examine the accuracy of using MLPsim and this equation to estimate the CPI of a configuration which may not be implemented in a cycle-accurate simulator. For this, we estimate the CPI of a configuration by substituting its MLP and MissRate measured by MLPsim and its CPI_{perf} and $Overlap_{CM}$ measured by the cycle-accurate simulator for *another* configuration (shown in normal font in Table 4), and we compare this to its CPI measured by the cycle-accurate simulator (shown in *italics* in Table 4).

The experiments were performed with Issue Window and ROB size = 64 and MissPenalty = 1000 cycles. The results show that in all cases, the estimated CPI using MLPsim is within 2% of the CPI measured using the cycle simulator.

5.3 Traditional Microarchitecture Features

In this section, we quantify the impact of traditional microarchitecture features such as instruction issue constraints, branch prediction and L2 cache size on achievable MLP.

5.3.1 Impact of Out-of-Order Instruction Issue

To study the impact of ROB size and out-of-order instruction issue constraints on MLP, we varied the ROB size from 16 to 256, and modeled five configurations of progressively aggressive issue constraints, as shown in Table 2. In this experiment, we set the issue window size to be equal to the ROB size.

The results of the experiment are shown in Figures 4 and 5. The graphs in Figure 4 plot MLP as a function of ROB/issue window size and issue constraints. Each curve on these graphs represents an issue configuration. The graphs in Figure 5 show the relative frequency of the different conditions that prevent more MLP from being achieved in an epoch. These frequencies are aver-

ROB/Issue Window Size	Issue Config	Database				SPECjbb2000				SPECweb99			
		Estimated using Config A	Estimated using Config B	Estimated using Config C	Measured	Estimated using Config A	Estimated using Config B	Estimated using Config C	Measured	Estimated using Config A	Estimated using Config B	Estimated using Config C	Measured
64	A	**7.97**	7.99	7.91	*8.02*	**2.91**	2.90	2.86	*2.91*	**2.38**	2.37	2.33	*2.37*
	B	7.81	**7.83**	7.75	*7.83*	2.86	**2.85**	2.81	*2.85*	2.37	**2.37**	2.33	*2.36*
	C	7.33	7.36	**7.28**	*7.28*	2.86	2.85	**2.81**	*2.80*	2.35	2.34	**2.31**	*2.30*

Table 4: Comparison Between Estimated and Measured CPI.

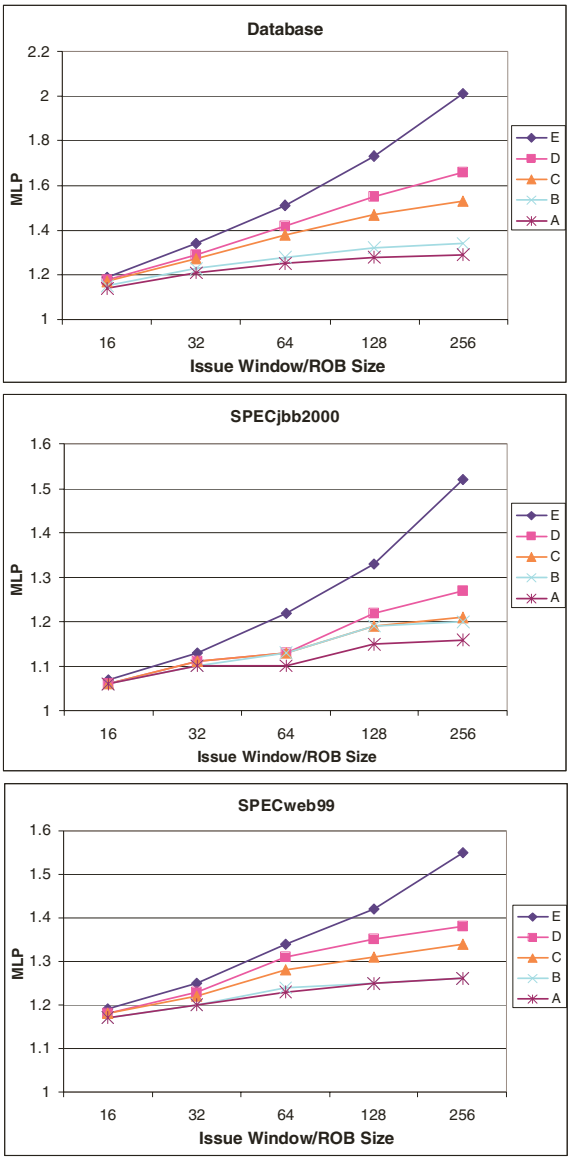

Figure 4: Impact of ROB Size and Issuing Constraints.

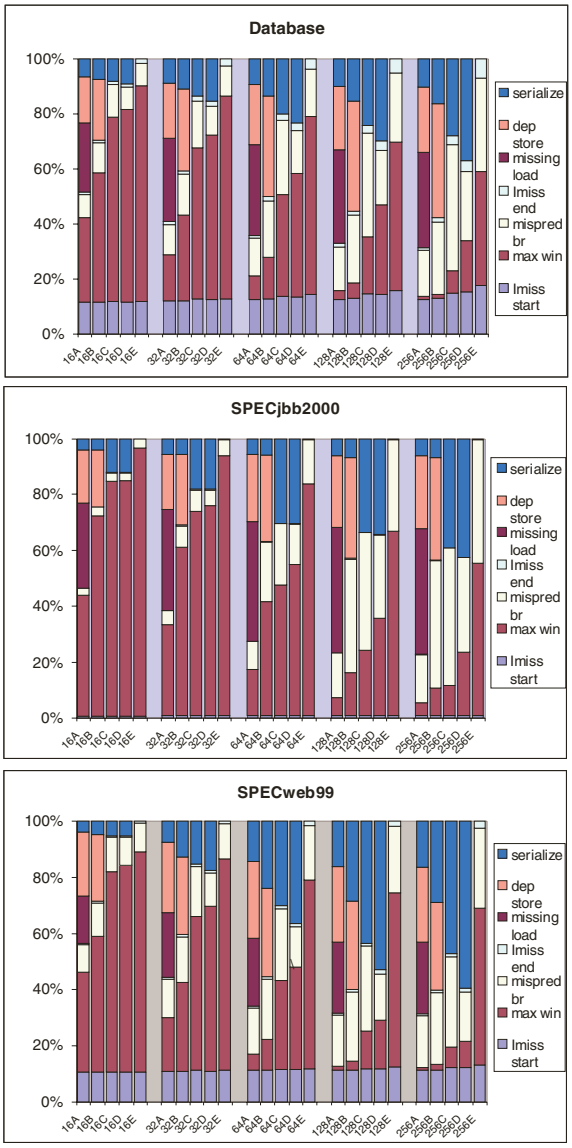

Figure 5: Factors Inhibiting Further MLP.

aged from all epochs.

As a basis for comparison, the MLP achieved by in-order instruction issue is shown in Table 5. The MLP achieved by in-order issue is somewhat higher in SPECweb99 because this benchmark contains a significant number of useful software prefetches. In general, stall-on-use only achieves marginally more MLP than stall-on-miss. Comparing the MLP achieved by our default processor configuration (i.e. "64C") to that achieved by an in-order stall-on-use processor, our moderately aggressive out-of-order processor achieves an MLP increase of 30% for the database workload, 12% for SPECjbb2000 and 13% for SPECweb99.

Benchmark	Stall-on-Miss	Stall-on-Use
Database	1.02	1.06
SPECjbb2000	1.00	1.01
SPECweb99	1.10	1.13

Table 5: MLP of In-Order Issue.

Examining Figure 4, we notice two major trends. Firstly, the database workload achieves the highest MLP among the three workloads as expected given that it has the highest L2 miss rate. Surprisingly though, SPECweb99, which has a miss rate equal to one tenth that of the database workload, also achieves good MLP, due to the strong clustering of misses in this benchmark, as discussed in Section 2.5, and the lack of dependencies between these misses. Secondly, as the sizes of the instruction issue window and ROB are increased, relaxing issue constraints becomes critical to achieving higher MLP. Most notably, at larger instruction issue window and ROB sizes, the serializing constraints of serializing instructions are a very serious impediment to exploiting MLP, especially for SPECjbb2000. Executing loads out-of-order benefits SPECjbb2000 when the ROB size exceeds 32, but it does not benefit the database workload and SPECweb99 unless loads are also allowed to speculate past earlier stores. For all three workloads, allowing branches to execute out-of-order becomes important when ROB size reaches 128 and beyond.

In Figure 5, each segment of a bar represents a condition that prevents more MLP from being uncovered in an epoch. *Imiss start* occurs when the epoch trigger is a missing instruction fetch. Since instruction fetch is blocking, no other off-chip accesses can be overlapped. *Maxwin* occurs when the issue window or the ROB is full. *Mispred br* occurs when a mispredicted branch that is dependent on a missing load is encountered. *Imiss end* occurs when the epoch trigger is a missing load or a missing prefetch and a missing instruction fetch prevented subsequent off-chip accesses from being overlapped. *Missing load* occurs when a missing load prevented subsequent missing loads or missing prefetches from being overlapped. This constraint is only valid under issue configuration A. *Dep store* occurs when a store that was unable to execute because of its address dependence on a missing load prevents subsequent missing loads or missing prefetches from being overlapped. This constraint is only valid for configurations A and B. *Serialize* occurs when a serializing instruction prevents subsequent missing loads and missing prefetches from being overlapped. The X-axis label in each graph refers to the issue window/ROB size and the issue configuration.

The results in Figure 5 show that missing instruction fetches (*Imiss start*) account for approximately 12-18% of all epoch triggers for the database workload and 10-13% of all epoch triggers for SPECweb99. We note that the latencies of these missing instruction fetches are fully exposed and highly detrimental to MLP and overall performance. Therefore, effective instruction prefetching for SPECweb99 and the database workload can improve MLP and overall performance significantly.

The results also show that for issue configurations A to D, beyond a window size of 32 instructions, *Maxwin* only accounts for 50% or less of the MLP inhibiting conditions. Therefore, issue window/ROB size limitation itself is only one of several impediments to achieving higher MLP. As noted above, for large issue window/ROB sizes, the serializing constraint of serializing instructions is actually the most serious impediment.

5.3.2 Decoupling Issue Window and ROB Size

In the previous section, we set the ROB size and the issue window size to be equal. In reality, the ROB is much easier to expand than the issue window because it is a FIFO buffer while the latter is a content addressable memory (CAM) structure. Making the issue window very large is unrealistic because it impacts the processor's cycle time. In this section, we study the MLP impact of making the ROB several times larger than the issue window. In Figure 6, each bar on a graph represents an issue window size and issue constraints configuration. For example, "16D" refers to an issue window size of 16 and issue configuration D. Each segment on a bar (except the segment labelled "INF") represents a ROB size that is some multiple of the issue window size. For example the "4X" segment on the "16D" bar refers to a ROB size of 64. The "2048" segment on the bar refers to a constant ROB size of 2048. The rightmost bar on each graph, labelled "INF" refers to a processor configuration with issue window size = 2048, ROB size = 2048 and issue configuration = E.

The results show that enlarging the ROB greatly improves MLP for the SPECjbb2000 and the database workload. The improvement increases with the more aggressive issue configurations and is dramatic for issue configuration E. For SPECweb99, enlarging the ROB does not show much improvement for issue configurations A to D but shows great improvement for issue configuration E.

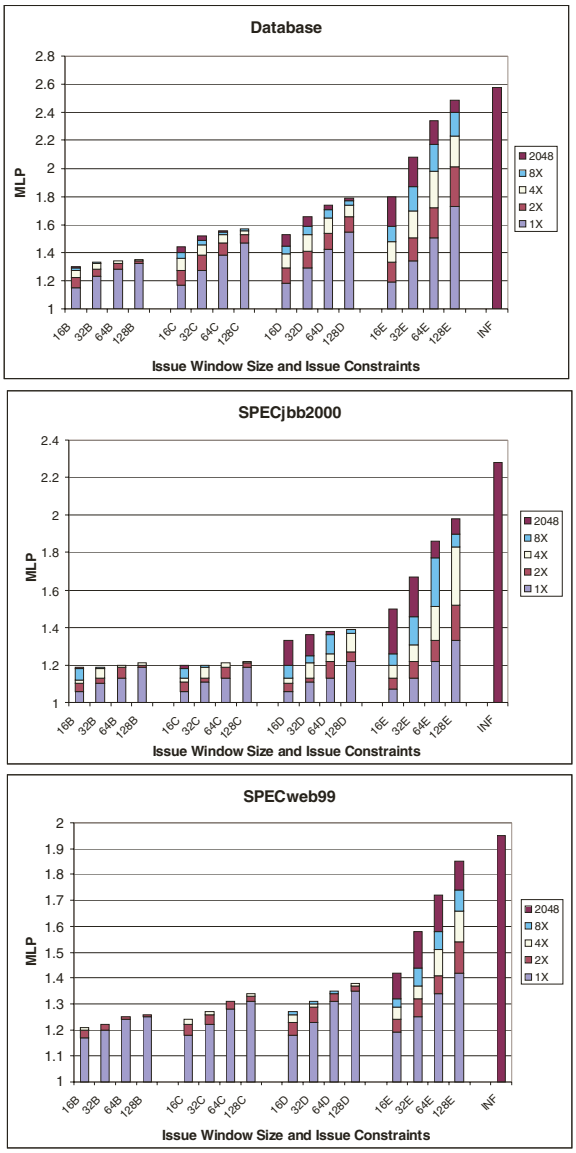

Figure 6: Impact of Decoupling Issue Window and ROB Sizes.

As an example of the MLP improvements attainable by an enlarged ROB, when the ROB for the "64D" processor configuration is enlarged from 64 to 256 entries, MLP improves by 16% for the database workload, by 12% for SPECjbb2000 and by 2% for SPECweb99. When the ROB for the "64E" configuration is enlarged from 64 to 1024 entries, the MLP improvement is 51% for the database workload, 49% for SPECjbb2000 and 22% for SPECweb99.

Overall, the results show that decoupling reorder buffer size from issue window size allows much more effective and efficient exploitation of MLP. The benefits of this decoupling has also been discovered by other researchers [19, 20].

5.3.3 Impact of Cache Sizes

Figure 7 shows the impact of L2 cache size on MLP. We

assume that misses to the L2 cache result in long-latency off-chip accesses. As cache size is increased, the intuitive expectation is that MLP will be reduced because the reduction in cache misses will likely result in increased distances between off-chip accesses, making the exploitation of MLP more difficult. This expectation holds for the database workload and SPECjbb2000 but not for SPECweb99. For SPECweb99, the misses that are eliminated occur mostly in epochs with low MLP. In such a situation, the reduction in the number of epochs tracks the reduction in the number of misses, and MLP (which is equal to ratio of the number of misses to the number of epochs) increases.

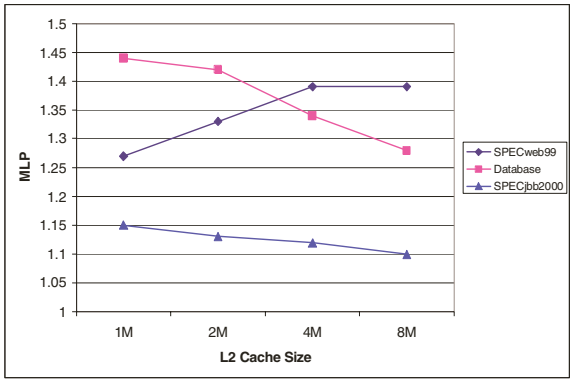

Figure 7: Impact of L2 Cache Size.

5.4 New Microarchitecture Features

In this section, we quantify the MLP impact of two recently proposed microarchitecture features: runahead execution[8, 9] and value prediction[10, 11, 12, 18].

5.4.1 Runahead Execution

As described in Section 3.5, runahead execution (RAE) increases MLP because it removes the issue window and reorder buffer size constraints of the processor as well as the serialization constraints of serializing instructions. Figure 8 compares the MLP

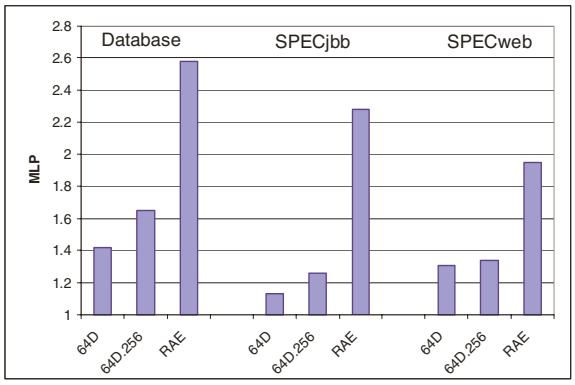

Figure 8: Impact of Runahead Execution.

achieved by runahead execution against two conventional out-of-order issue processor configurations. Both of these configurations have 64-entry issue windows and issue configuration D. They differ in that the first configuration has a 64-entry ROB while the second has a 256-entry ROB. In this experiment, we assume that the RAE processor can runahead up to at most 2048 instructions in RAE mode. In reality, the maximum runahead distance is dependent on the off-chip access latency.

The results show that runahead execution demonstrates impressive improvements in MLP over the two conventional out-of-order issue processor configurations. These improvements are 82% and 56% for the database workload, 102% and 81% for SPECjbb2000, and 49% and 46% for SPECweb99.

The observant reader will notice that the RAE results are identical to the "INF" (rightmost) bars in Figure 6. In effect, RAE is an alternative and much more realistic means of achieving the effects of a very large issue window and a very large ROB, and of removing the serializing constraints of serializing instructions. We also note that there are other microarchitecture ideas such as speculative precomputation [15], preexecution [16, 17] and others [21, 22, 23] that share the MLP-enhancing properties of runahead execution.

5.5 Value Prediction

Figure 9 shows the performance improvement obtained by adding value prediction to the same processor configurations shown in Figure 8. The value predictor used is a last-value predictor with 16K entries and is used to predict only the values of missing loads. Its accuracy and coverage are shown in Table 6.

Benchmark	Correct	Wrong	No Predict
Database	42%	7%	51%
SPECjbb2000	20%	3%	77%
SPECweb99	25%	5%	70%

Table 6: Value Predictor Statistics.

The results show that value prediction improves MLP of the database workload by 4%-9%, with the RAE configuration showing the most gain. For SPECjbb2000 and SPECweb99, the performance gain for the two conventional out-of-order processor configurations is negligible. The performance gain for RAE is 2% for SPECjbb2000 and 5% for SPECweb99. Arguably, missing load value prediction is only worthwhile when combined with RAE. Value prediction in RAE is also easier to implement since it does not require a recovery mechanism [7].

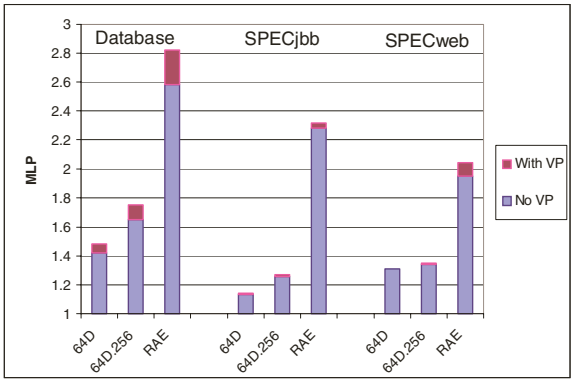

Figure 9: Impact of Value Prediction.

5.6 Limit Study

In this section, we attempt to find profitable research directions for improving MLP by performing a limit study assuming in turn perfect instruction prefetching, perfect value prediction, perfect branch prediction and the combination of perfect value prediction and perfect branch prediction.

The upper graph in Figure 10 assumes a baseline processor that implements RAE. It shows that for the database workload and SPECweb99, perfect instruction prefetching (RAE.perfI), perfect

value prediction (RAE.perfVP) and perfect branch prediction (RAE.perfBP) all show similarly good MLP gains. The gains are 39%-48% for the database workload and 21%-23% for SPECweb99. For SPECjbb2000, perfect instruction prefetching does not show any MLP gain (because missing instruction fetches are not a problem) but perfect value prediction and perfect branch prediction improves MLP by 56% and 45% respectively. When perfect value prediction is combined with perfect branch prediction (RAE.perfVP.perfBP), the MLP gain is 134% for the database workload, 215% for SPECjbb2000 and 57% for SPECweb99.

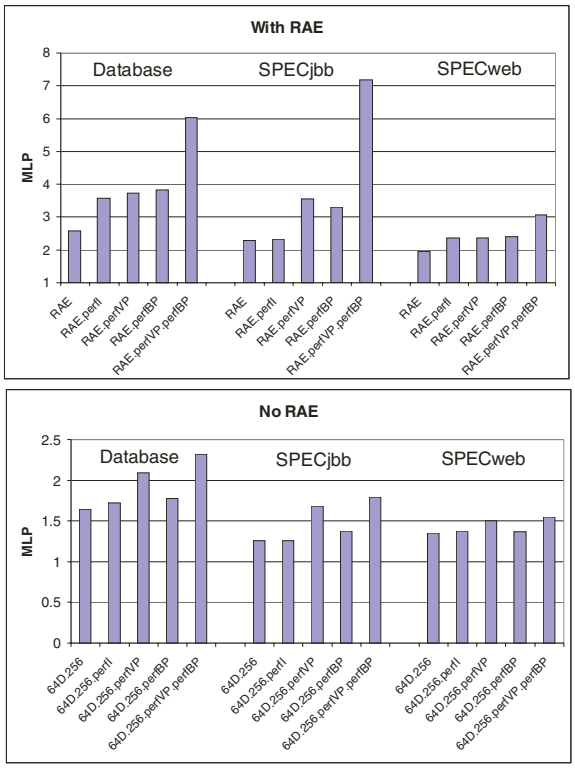

Figure 10: Impact of Perfect Instruction Fetch, Branch Prediction and Value Prediction.

The lower graph in Figure 10 assumes a baseline processor without RAE. This processor has a 64-entry issue window, 256-entry reorder buffer and implements issue configuration D. Compared to the processor with RAE, the MLP gains are modest.

These results shown that there is still considerable headroom in improving MLP beyond that attainable by RAE. In our opinion, improving instruction prefetching is probably easier than improving branch prediction and value prediction and is therefore the most promising avenue for further improving MLP for SPECweb99 and the database workload.

5.7 MLP and Overall Performance

So far, we have presented all our results in terms of MLP. We now relate these gains in MLP back to gains in overall performance. Figure 11 shows the percentage performance improvement of a sample of processor configurations studied in Sections 5.3-5.6 relative to the "64D" processor configuration. The CPI estimates for each configuration are obtained by substituting their MLP from MLPsim and other metrics from Table 1 into the second equation of Section 2.3. The off-chip access latency used is 1000 cycles.

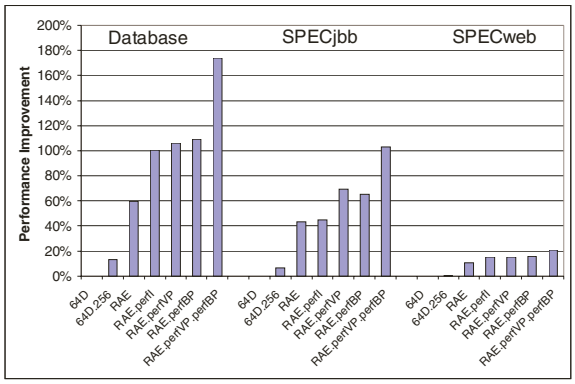

Figure 11: Overall Performance Improvement.

The results show that RAE improves overall performance by 60% for the database workload, by 44% for SPECjbb2000 and by 11% for SPECweb99. These are impressive performance gains. In the limit when runahead execution is combined with perfect branch prediction and perfect value prediction, the maximum performance improvement is 174% for the database workload, 103% for SPECjbb2000 and 21% for SPECweb99.

6 Related Work

As far as we know, the term MLP was first used by Glew [5], who qualitatively argued why microarchitects should focus on it rather than on ILP. Pai and Adve [6] proposed a software approach of using compiler code transformations to increase MLP in scientific applications. Zhou and Conte [7] recently proposed using value prediction to enhance MLP instead of ILP. In their study, they consider all cache misses as contributing to MLP, including those that hit in the L2 cache.

Sorin et al's [24] fM parameter appears to be similar to our definition of MLP but their parameter counts all memory accesses (both useful and useless) while our MLP metric counts only useful accesses. Our MLPsim simulator is similar to their FastILP simulators in some ways. For example, the notion of epochs in MLPsim is identical to the notion of eras in FastILP. However, unlike FastILP, MLPsim is completely timing-unaware and does not need to model the cycle behavior of on-chip computation. Pai et al's [25] and Ranganathan et al's [26] L2 MSHR occupancy are similar to Sorin's fM parameter in that they are simply counts of the number of outstanding memory accesses. The main focus of these three papers was on the modeling and performance evaluation of multiprocessor systems with ILP processors. In contrast, the focus of this paper is a detailed and systematic study of how microarchitecture affects achievable MLP.

7 Conclusion and Future Work

In summary, this paper makes the following contributions:

- We define MLP formally, show how it can be measured, and relate it to overall processor performance.
- We show that off-chip accesses are clustered in the commercial applications we studied, making the exploitation of MLP feasible.
- We use the epoch model to reason about MLP and how microarchitecture features qualitatively impact MLP.
- We show that out-of-order execution is moderately effective in exploiting MLP and quantify the impact of microarchitecture

parameters such as issue window size, reorder buffer size and on-chip cache sizes on MLP. We demonstrate that instruction issuing constraints have to be relaxed in order to attain the benefits of large issue windows and that serializing instructions are a major impediment to MLP.

• We show that decoupling the reorder buffer from the issue window and enlarging it increases MLP significantly.

• We demonstrate that runahead execution is highly effective in exploiting MLP because it removes the issue window and reorder buffer size constraints of the processor and because it removes the serialization constraints of serializing instructions.

• We show that missing load value prediction is somewhat effective in improving MLP when combined with runahead execution.

• Our limit study shows that there is very significant headroom in improving MLP via instruction prefetching, more accurate branch prediction and better value prediction.

• We show that for the database workload and SPECjbb2000, these improvements in MLP translate into impressive overall performance gains.

In conclusion, this study shows that microarchitecture features and parameters have a profound impact on achievable MLP and that exploiting MLP is an effective approach for improving the performance of memory bound applications. Our future work includes studying MLP for multithreaded processors as well as studying store MLP for applications where a finite store buffer limits performance. Based on the insights of this study, we also plan to study new microarchitecture ideas for further improving MLP.

Acknowledgment

We would like to thank Hui-May Chang, Peter Lawrence, Harit Modi, Khoa Nguyen, Tien-Pao Shih and Jhy-Chun Wang for their infrastructure contributions. We would also like to thank Craig Anderson, Sorin Iacobovici, Gurindar Sohi, Rabin Sugumar, Marc Tremblay and Stevan Vlaovic for reviewing early drafts of this paper.

References

[1] A. Maynard, C. Donelly and B. Olszewski, “Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads,” in ASPLOS-VI, 1998.

[2] L. Barroso, K. Gharachorloo, E. Bugnion, “Memory System Characterization of Commercial Workloads,” in 25th International Symposium on Computer Architecture, 1998.

[3] R. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eric, H. Nueckel and J. Shen, “Scaling and Characterizing Database Workloads: Bridging the Gap between Research and Practice,” in 36th International Symposium on Microarchitecture, December 2003.

[4] W. Wulf, and S. McKee, “Hitting the Memory Wall: Implications of the Obvious,” in Computer Architecture News, Vol. 23, No. 4, September 1995.

[5] A. Glew, “MLP yes! ILP no!,” in ASPLOS Wild and Crazy Idea Session ‘98, October 1998.

[6] V. Pai and S. Adve, “Code Transformations to Improve Memory Parallelism,” in 32nd International Symposium on Microarchitecture, November 1999.

[7] H. Zhou and T. Conte, “Enhancing Memory Level Parallelism via Recovery-Free Value Prediction,” in International Conference on Supercomputing, June 2003.

[8] J. Dundas and T. Mudge, “Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss,” in International Conference on Supercomputing, July 1997.

[9] O. Mutlu, J. Stark, C. Wilkerson and Y. Patt, “Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors,” in 9th International Sysmposium on High Performance Computer Architecture, February 2003.

[10] M. Lipasti and J. Shen, “Value Locality and Load Value Prediction,” in ASPLOS-VII, October 1996.

[11] F. Gabbay and A. Mendelson, “Speculative Execution Based on Value Prediction,” in EE Department Tech Report 1080, Technion - Israel Institute of Technology, November 1996.

[12] Y. Sazeides and J. Smith, “The Predictability of Data Values,” in 30th International Symposium on Microarchitecture, 1997.

[13] D. Weaver and T. Germond, “The SPARC Architecture Manual,” PTR Prentice Hall, 1994.

[14] www.spec.org

[15] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery and J. Shen, “Speculative Precomputation: Long-Range Prefetching of Delinquent Loads,” in 28th International Symposium on Computer Architecture, 2001.

[16] C. Luk, “Tolerating Memory Latency Through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors,” in 28th International Symposium on Computer Architecture, 2001.

[17] D. Kim and D. Yeung, “Design and Evaluation of Compiler Algorithms for Pre-Execution,” in ASPLOS-X, October 2002.

[18] K. Wang and M. Franklin, “Highly Accurate Data Value Prediction Using Hybrid Predictors,” in 30th International Symposium on Microarchitecture, November 1997.

[19] T. Karkhanis and J. Smith, “A Day in the Life of a Data Cache Miss,” in Workshop on Memory Performance Issues, May 2002.

[20] H. Akkary, R. Rajwar and S. Srinivasan, “Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors,” in 36th International Symposium on Microarchitecture, December 2003.

[21] A. Roth and G. Sohi, “Speculative Data-Driven Multithreading,” in 7th International Symposium on High-Performance Computer Architecture, January 2001.

[22] A. Moshovos, D. Pnevmatikatos and A. Baniasadi, “Slice-Processors: An Implementation of Operation-Based Prediction,” in International Conference on Supercomputing, June 2001.

[23] M. Dubois and Y. Song, “Assisted Execution,” University of Southern California CENG Technical Report 98-25, 1998.

[24] D. Sorin et al, “Analytic Evaluation of Shared-Memory Systems with ILP Processors,” in 25th International Symposium on Computer Architecture, 1998.

[25] V. Pai, P. Ranganathan and S. Adve, “The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology,” in International Symposium on High Performance Computer Architecture, February 1997.

[26] P. Ranganathan, K. Gharachorloo, S. Adve and L. Barroso, “Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors,” in ASPLOS-VIII, 1998.